NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DEADLOCK DETECTION

IN DISTRIBUTED COMPUTING SYSTEMS

by

Michael T. Gehl

June 1982

Thesis Advisor: Dushan Z. Badal

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): DEADLOCK DETECTION IN DISTRIBUTED COMPUTING SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1982 / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Michael Thomas Gehl	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June 1982 / 13. NUMBER OF PAGES: 74
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Distributed Computing Systems, Concurrency Control, Deadlock, Distributed Database Systems, Computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

With the advent of distributed computing systems, the problem of deadlock, which has been essentially solved for centralized computing systems, has reappeared. Existing centralized deadlock detection techniques are either too expensive or they do not work correctly in distributed computing systems. Although several algorithms have been developed specifically for distributed systems, the majority of them have also been shown

20. (Continued)

to be inefficient or incorrect. Additionally, although fault-tolerance is usually listed as an advantage of distributed computing systems, little has been done to analyze the fault tolerance of these algorithms. This thesis analyzes four published deadlock detection algorithms for distributed computing systems with respect to their performance in the presence of certain faults. A new deadlock detection algorithm is then proposed whose efficiency and fault tolerance are adjustable.

Approved for public release; Distribution unlimited

Deadlock Detection in Distributed Computing Systems

by

Michael T. Gehl
Lieutenant Commander, United States Navy
B.S., Iowa State University, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

ABSTRACT

With the advent of distributed computing systems, the problem of deadlock, which has been essentially solved for centralized computing systems, has reappeared. Existing centralized deadlock detection techniques are either too expensive or they do not work correctly in distributed computing systems. Although several algorithms have been developed specifically for distributed systems, the majority of them have also been shown to be inefficient or incorrect. Additionally, although fault-tolerance is usually listed as an advantage of distributed computing systems, little has been done to analyze the fault tolerance of these algorithms. This thesis analyzes four published deadlock detection algorithms for distributed computing systems with respect to their performance in the presence of certain faults. A new deadlock detection algorithm is then proposed whose efficiency and fault tolerance are adjustable.

TABLE OF CONTENTS

I. INTRODUCTION

A. STATEMENT OF THE PROBLEM

Deadlock is a circular wait condition which can occur in any multiprogramming, multiprocessing or distributed computer system which uses Locking to maintain consistency of the data base, if resources are requested when needed and processes are not assigned priorities. It indicates a state in which each member of a set of transactions is waiting for some other member of the set to give up a lock. An example of a simple deadlock is shown in Figure 1. Transaction T1 holds a lock on resource R1 and requires resource R2; transaction T2 holds a lock on resource R2 and requires R1. Neither transaction can proceed, and neither will release a lock unless forced by some outside agent.

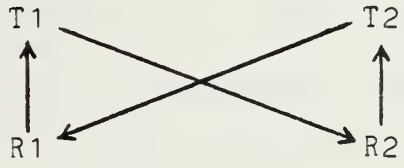


Fig. 1 -- A simple deadlock cycle

There have been many algorithms published for deadlock detection, prevention or avoidance in centralized multiprogramming systems. The problem of deadlock in those

systems has been essentially solved. With the advent of distributed computing systems, however, the problem of deadlock reappears. Certain peculiarities of distributed systems (lack of global memory and non-neglibible message delays, in particular) make centralized techniques for deadlock detection expensive and incorrect in the sense that they do not detect all deadlocks and/or they detect false deadlocks. Although there have been several deadlock detection algorithms for distributed systems published, most of them have been shown to be incorrect.

B. PROPOSED SOLUTION TO THE PROBLEM

In this thesis, a new deadlock detection algorithm for distributed computing systems is proposed which is low cost in terms of inter-site messages. The proposed algorithm is also able to be dynamically modified to make it more robust. The algorithm assumes a model of transaction execution wherein a transaction which requires a resource located at another site will "migrate" to that site to utilize the resource. The major differences between the proposed algorithm and existing algorithms are the concept of a Lock History which each transaction carries with it, and a three staged approach to deadlock detection, with each stage, or level, of detection activity being more complex than the preceding.

C. STRUCTURE OF THE THESIS

In Chapter two, a more detailed discussion of the deadlock problem and published solutions is presented. For a reader with little background in this problem, it presents a brief introduction to the published literature on the deadlock condition. A reader more familiar with the condition of deadlock may wish to proceed directly to Chapter three or four. Chapter three presents an analysis of four published algorithms with respect to their robustness in the presence of single site failures and lost messages between sites. The four algorithms are executed on the same example so that they can be easily compared.

In Chapter four, one version of the proposed algorithm is presented. This version is the least robust version available, and it is presented for ease of comparison with existing algorithms. An informal proof of correctness and a comparison with the algorithm of Obermarck [Ref. 1] is included in Chapter five. The algorithm is also executed on the example of Chapter three for comparison of its robustness with that of the algorithms analyzed in that chapter. Chapter six discusses several modifications which can be made to the proposed algorithm to increase its robustness. The conclusions reached by the author during this research are also presented in Chapter six.

II. THE CONDITION OF DEADLOCK

In the past decade there has been considerable work done on distributed computer networks and multiprocessor systems. Both of these are predecessors of distributed computing systems which are presently a focus of intensive research and development in academia and industry. Many techniques for concurrency control, reliability, recovery or security developed for centralized (or single CPU) systems have been or are being adopted and adapted for distributed computing systems. For example, there is a tendency to use locking as a general synchronization technique in distributed systems and its special variant, two-phase locking, for distributed database systems. Up until recently it has been argued that the frequency of deadlock occurence in existing applications is so low that the problem of deadlock in distributed systems is not very important and therefore can be managed by adopting techniques developed for centralized systems. However, it has become recently apparent that deadlocks may be a problem in the future as new applications featuring large processes and/or many concurrent processes or transactions arise [Ref. 2]. An example of such an application is an information utility system which services concurrently hundreds or perhaps thousands of TV users.

Distributed computing systems are characterized by the absence of global memory and by message transmission delays which are not negligible. Additionally, processes operating at the same or different sites can communicate with each other, and can share resources. If locking is used as the synchronization technique, then the last two items raise the problems of deadlock occurence in distributed systems, and the first two characteristics of distributed systems make it much more difficult to detect, avoid or prevent deadlock than in the earlier multiprogramming centralized computing systems.

Deadlock prevention and avoidance algorithms for a distributed computing systems are generally not efficient. Prevention can be accomplished by 1) not allowing concurrent processing, 2) assigning priorities and allowing preemption, 3) requiring a process to acquire all resources it will need before it starts, or 4) having no locks. Requiring sequential execution in a distributed system is a gross waste of resources. Having prioritized processes will result in lower-prioritied processes being restarted many times, with a major degradation in system efficiency. Dynamic prioritization would be a complex and time consuming algorithm by itself. A process may be unable to determine its minimum set of resources, and therefore would have to acquire the set of all probable and possible resources, even though it may not need them. In addition, in systems in

which messages are treated as resources, it is impossible to determine in advance which messages will be required. Assuming a non-optimistic concurrency controller, having no locks may result in database inconsistencies. Similarly, deadlock avoidance algorithms, which either calculate a "safe path" [Ref. 3] or never wait for a lock [Ref. 4] are also inefficient. Safe path algorithms require a non-trivial execution time, and must be done each time a resource request is to be granted. Never waiting for a lock is inefficient when deadlock is a rare occurence. Thus, in distributed computing systems, it appears that deadlock detection and resolution algorithms should be investigated to determine if they are a more efficient method of handling deadlock.

There are four criteria that any deadlock detection algorithm for distributed computing systems must meet. They are 1) correctness, 2) robustness. 3) performance, and 4) practicality. Correctness refers to the ability of the algorithm to detect all deadlocks, and the ability to not detect any false deadlocks. Robustness refers to the ability of the algorithm to be correct even in the presence of anticipated faults. This includes the ability to detect deadlocks even when a site fails or loses communications while the deadlock detection algorithm is being executed. The performance of the algorithm refers to its overhead -- the delays between deadlock and detection, CPU time used,

number of messages required, etc. Practicality is closely related to performance, and refers to aspects such as complexity and cost.

The two major approaches to deadlock detection and resolution are centralized and distributed deadlock detection algorithms. Within the distributed class are two subclasses; 1) all or several sites execute the deadlock detection algorithm, and 2) only one site is actually executing, although the algorithm is resident in all sites and thus any site could execute the algorithm. It might be easier to view the algorithms as a continuum: fully centralized [Ref. 4], hierarchical [Ref. 5], distributed with a single site at a time executing the algorithm [Ref. 3], distributed with all sites involved in a possible deadlock executing the algorithm concurrently [Ref. 5], and distributed with all sites executing the algorithm concurrently [Ref. 6].

The robustness of several published deadlock detection and resolution algorithms for distributed systems will be analyzed in Chapter three. The motivation for this analysis comes from three facts. First, very few authors investigated robustness or reliability of deadlock detection algorithms. Second, reliable deadlock detection and resolution for upcoming new distributed systems and applications is an urgent, very important and as yet not satisfactorily resolved problem. Third, as there can be

more than one deadlock being detected by the deadlock detection algorithm, it is reasonable to expect such an algorithm to be robust, i.e., to continue executing and detecting all deadlocks even in the presence of failure(s) which might have in effect broken one of the deadlocks being detected.

The analysis of the robustness of the deadlock detection algorithms (DDA) will concentrate on investigating the impact of some single failures on such algorithms. In general, the DDA is invoked by two events - either whenever a process waits for a resource, or after a certain period of time has elapsed since the last DDA invocation. In the first case, deadlock is checked for whenever its possibility appears, and in the second case it is checked for periodically (regardless of whether its possibility exists). A variant of the first case is to delay checking for deadlock for some period of time on the premise that most transaction waits are transitory and will not become deadlocked.

The DDA can reside in one, several or all sites of the distributed computing system. When a triggering event for DDA occurs, one, several or all sites, depending on the particular algorithm, will receive information from several or all sites. Such information consists of "who waits for whom and where", and it can be represented by arcs of the wait-for graph, strings, or lists of processes or

transactions. Upon receipt of such information one, several or all sites attempt to reconstruct a global state of the distributed system, i.e., to generate a true snapshot either of all waiting processes or of all processes in the system.

The generation of such a true snapshot in the distributed system is difficult and perhaps even impossible because of message delays and the lack of global memory. What is desired, however, is a true snapshot of the deadlock cycle: the status of other transactions in the system should be inconsequential to the deadlock detection process. The generation of such a true snapshot of the deadlock cycle, usually referred to as a global wait-for graph, becomes more difficult when we consider the possibility of failures in the distributed system. Some system mechanisms have been designed to be robust or reliable. For example, some concurrency control or synchronization mechanisms for distributed databases and transaction processing systems are based on two phase locking, which has been made robust by incorporating atomicity by using two phase commit protocols. The two phase commit protocol supports not only the atomicity of transactions but it also supports the robustness of locking, i.e., the robustness of concurrency control mechanisms. In particular what makes the concurrency control which uses locking robust is the need to lock and unlock resources in a robust way, i.e., either all lock/unlock operations for a given process or transaction

occur or none occur. Thus the robustness of concurrency control supports the atomicity of placing and releasing a set of locks needed by a process. In other words, robustness of concurrency control means that no dangling locks or locked resources are left behind the terminated or committed process, even in the presence of some failures. It is interesting to note that although deadlock detection is a part of concurrency control based on locking, there has been no attempt to provide for or even to investigate the robustness of deadlock detection mechanisms. The most likely explanation for this is that from the concurrency control point of view, the inability of the process to lock a needed resource is an exception to be handled by another mechanism, a deadlock detection algorithm (DDA).

The proper way to see the DDA is as another transaction running under the concurrency control mechanism, as it reads and shares lock tables with concurrency controllers and other transactions. However, the DDA is a special transaction which operates on special data it creates solely for deadlock detection, e.g., wait-for graphs. This data, called deadlock data, is internal to each invocation of a DDA transaction and is erased after its execution. Moreover, deadlock data is not shared by any other DDA transaction invocations and therefore need not be locked. This means that the robustness required of DDA transactions is of a somewhat different kind than the robustness of

transactions operating on shared database data. The DDA transaction therefore does not need to use a two phase commit protocol to ensure its robustness.

Consider the following informal model of DDA transaction execution. The DDA is invoked by a concurrency controller at a site at which a database transaction can not acquire locks which are being held by another transaction(s). The DDA transaction executes at one, several or all sites (depending on the DDA itself and the deadlock topology). During its execution the DDA transaction should exhibit the atomicity property, i.e., it either executes correctly or it does not execute at all. The results of DDA transaction execution is either of two messages to the concurrency controller which triggered it:

1) Proceed - because of
   a) no deadlock
   b) deadlock detected but another transaction was selected as a victim
2) Abort - because of
   a) deadlock detected and you are the victim.
   b) DDA transaction failed

In Chapter three, two classes of single failures will be considered. First, the impact of lost messages will be analyzed and second, the impact of one site failures or one site partitions on DDA behavior will be analyzed. The impact of lost messages is analyzed because 1) not all distributed systems support reliable delivery of messages, 2) several algorithms treat messages as resources [Ref. 3], and 3) in some applications acknowledgements cannot be sent.

III. ANALYSIS OF DEADLOCK DETECTION ALGORITHMS

In this chapter, four published deadlock detection algorithms for distributed computing systems are examined with respect to the presence of the two classes of failures (lost messages and site failures) discussed in Chapter two. Although very few of them have already been shown to be correct when no failures or errors occur, their robustness is nevertheless worth analyzing. The assumptions made by each author will be discussed in the context of how robust the algorithm is. Each DDA will be analyzed by executing it in the following environment.

There are four sites in the system, each of which has a single resource and a single transaction. (These restrictions merely make the example simpler, they are not required for the analysis.) The initial system status is shown in Figure 2. Transaction T1 at site A holds resources R2 and R3 and is waiting for resource R4. Transactions T2 and T3 hold no resources. Transaction T4 at site D holds resource R4, and is active. It is assumed that the deadlock detection activity resulting from T1 waiting for R4 has been completed, so there is currently no deadlock detection activity in the system. An arrow from one transaction to another indicates that the first transaction is waiting for the second transaction to release a lock. An arrow from a

resource to a transaction indicates that the transaction has a lock on that resource, while an arrow from a transaction to a resource indicates that the transaction desires to put a lock on that resource. For the algorithms which require global timestamps, timestamp (TS) t1 is assigned to the T1<--R2 assignment, t2 to the T4<--R4 assignment, t3 to the T1<--R3 assignment, and t4 to the T1-->R4 request. Now at some time t6, transaction T4 requests R3, resulting in a global deadlock T1-->T4-->T1.

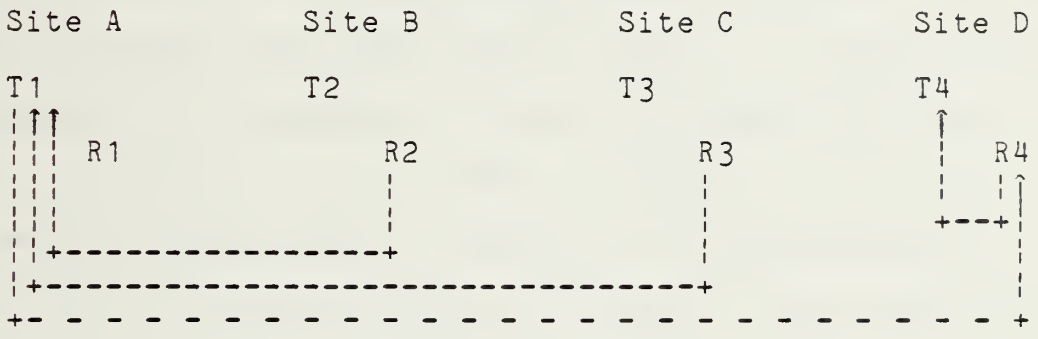


Fig. 2 -- Initial status of deadlock example

In the case of a site failure, the following possibilities may exist. a) A site can have a transaction involved in a deadlock but not be involved in deadlock detection, b) a site can have a transaction involved in a deadlock and be involved in detection, c) a site can have a resource involved in a deadlock and not be involved in detection, d) a site can have a resource involved in a deadlock and be involved in a detection, or e) a site can be involved in deadlock detection but in no way involved in a deadlock.

A. THE ALGORITHM OF GOLDMAN

In [Ref. 3], Goldman presents two deadlock detection algorithms. Only the distributed version will be considered in this paper. A Process Management Module (PMM) at each site handles resource allocation and deadlock detection. An "ordered blocked process list" (OBPL) is a list of process names, each of which is waiting for access to a resource assigned to the preceeding process in the list. The last process in the list is either waiting for access to the resource named, or it has access to that resource. An OBPL is created each time a PMM wants to see if a blocked process is involved in a deadlock. In the distributed algorithm, an OBPL is passed from a PMM to another PMM which has information either about a resource or a transaction in the OBPL which is needed to expand the OBPL. Each PMM adds the information it knows, and either detects a deadlock. detects a non-deadlocked state, or passes the OBPL to another PMM for further expansion. The terms process and transaction will be used synonymously in the analysis of this DDA. If several transactions are waiting on one transaction, multiple copies may be made of the OBPL and sent to each site having one of those waiting transactions. Processes can be in either of 2 states, active or blocked (waiting). A blocked process could be waiting for a database object, message text from another process or message text from an operator. A process is active if it is not blocked. In the

algorithm, PX and RX are temporary variables representing a process or resource. The steps of the algorithm are:

1. Set RX to the value contained in the resource identification portion of the OBPL. If RX represents a local resource, go to 2. Otherwise, go to 8.

2. Verify that the last process added to OBPL is still waiting for RX. If so, go to 3, otherwise, halt.

3. Let PX be process controlling RX. If PX is already in OBPL, then there is a deadlock. If not, go to 4.

4. If PX is local to current PMM, go to 5, otherwise go to 7.

5. If PX is active, there is no deadlock. Discard OBPL and halt. Otherwise go to 6.

6. Add PX to OBPL and go to 10.

7. Add PX and RX to OBPL. Send OBPL to PMM in site in which PX resides. Halt.

8. Verify that last process in OBPL still has access to RX. If not, there is no deadlock, so discard OBPL and halt. If so, go to 9.

9. If last process in OBPL is active, there is no deadlock, so discard OBPL and halt. Otherwise go to 10.

10. Call resource for which last process is waiting RX. If RX is local, go to 3. Otherwise go to 11.

11. Place RX in OBPL and send OBPL to PMM of site in which RX resides. Halt.

Figure 3 shows the actions taken at each site during the execution of the DDA following the request by T4 for resource R3. The numbers refer to the current step being

executed by the DDA. As can be seen, the algorithm correctly detected the resulting deadlock, in an environment of no faults. If, however, a message is lost (in this example, either the OBPL sent from site C to A, or the OBPL sent from A to D), the necessary information to detect the deadlock will be lost, and the algorithm will fail to detect an existing deadlock.

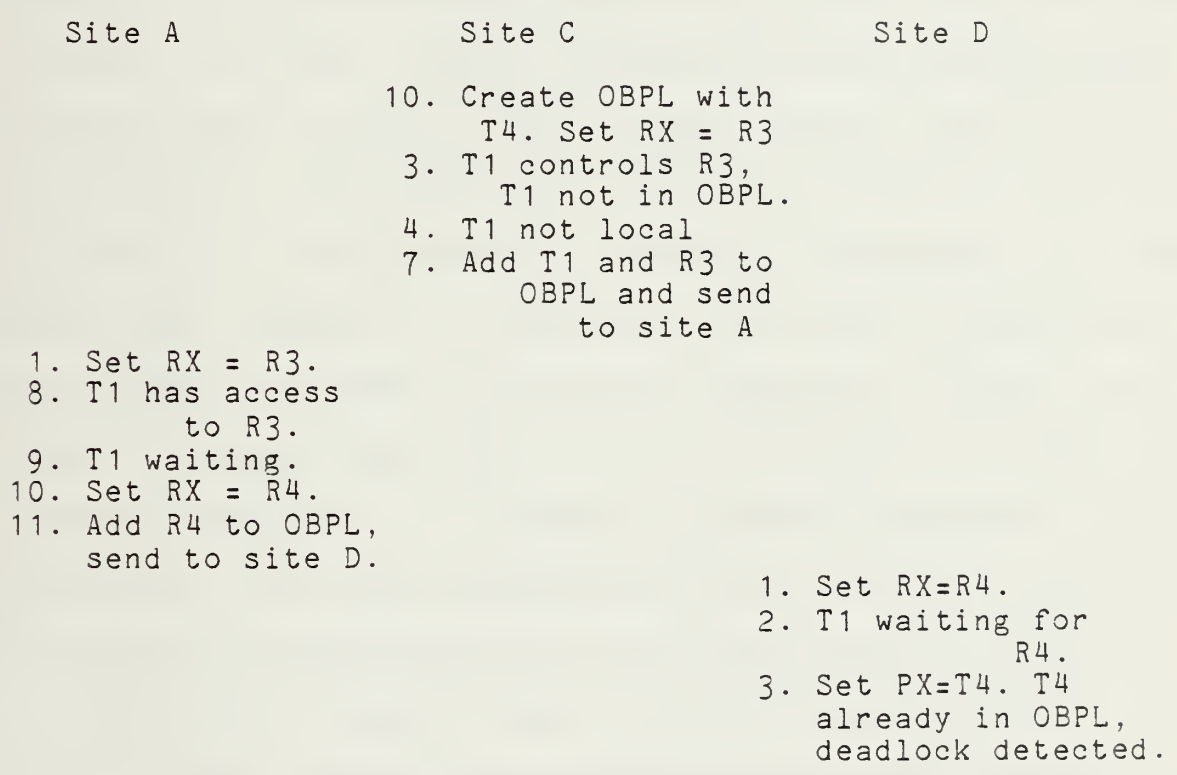


Fig. 3 -- Execution of the Goldman DDA

Goldman's algorithm allows the following types of sites discussed previously: type b (a site can have a transaction involved in deadlock and the site is involved in detection), type d (a site can have a resource held by a transaction involved in deadlock and the site will be involved in

deadlock detection), and type c (a site can have a resource held by a transaction involved in a deadlock and not be involved in deadlock detection). A site could also be in several of the categories above, depending on the complexity of the system state. For example, site D could be considered a type b or type d site. If a site of type b (sites A or D in this example) fails during execution of the DDA, the behavior could be different depending on the time of the failure. If the failure occured at site A before site C sent the OBPL to site A, site C would realize that site A had failed. The algorithm includes no procedure for this occurence, so the behavior would be dependent on the underlying system. If the failure at site A occured after it received the OBPL, all deadlock detection activity will cease, because only site A was currently involved in deadlock detection. A system timeout mechanism would eventually abort the transactions involved in the deadlock. A failure at site D would have the same effect as at site A.

If a site of type d (site C in this example) failed, the time of the failure would again determine the behavior of the DDA. If the failure occured before site C sent the OBPL to site A, deadlock detection activity would cease without deadlock having been detected. If the OBPL had been sent, however, deadlock detection would continue at sites A and D (sequentially) with site D detecting a deadlock. The failure of site C would not have been critical after the

OBPL had been sent. The effect of a type c site (site B in this example) failing would have no effect on the behavior of the DDA, because the fact that R2 is held by T1 is not used or known by the DDA at any site.

There are essentially two types of OBPL's created by this DDA. The first type is when a process is waiting, but is not involved in a deadlock. This OBPL is subsequently discarded. The second type is one which will eventually show a deadlock cycle. If there are n transactions involved in a deadlock cycle, this DDA will create from 1 to n OBPL's. In this example, only one was created. If the request by T1 for resource R4 hapened simultaneously with the request by T4 for resource R3, two OBPL's would have been created which would have resulted in two sites independently detecting the same deadlock, vice the one site in this example. Thus the robustness of this algorithm with respect to a single site failure is related to the ratio of the number of OBPL's created to the number of transactions involved in the deadlock. This ratio is determined by the sequencing or timing of transactions requesting blocked resources which is of a random nature. A ratio of 1 would provide the highest degree of robustness. When only a single OBPL is created, the robustness of the DDA is very similar to that of a centralized DDA; a single site failure can stop deadlock detection activity. The robustness of this DDA can therefore be analyzed but not predicted.

B. THE ALGORITHM OF MENASCE-MUNTZ

In [Ref. 5], Menasce and Muntz presented a distributed deadlock detection algorithm. Gligor and Shattuck [Ref. 7] presented a counter example which showed the algorithm to be incorrect in that it failed in some cases to detect a deadlock. They also proposed a modification to the algorithm which they thought would make it correct, but they felt the algorithm was impractical. In [Ref. 8], Tsai and Belford show that the algorithm as modified by Gligor and Shattuck is also incorrect.

The algorithm constructs a Transaction-Waits-For (TWF) graph at originating sites of transactions which are potentially involved in the deadlock being detected, and at sites at which some transaction could not acquire a resource. Nodes in the WF graphs represent transactions. An edge (Ti,Tj) indicates that transaction Ti is waiting for transaction Tj. A non-blocked transaction is a transaction that is not waiting and is represented in the TWF graph by a node with no outgoing arcs. A blocked transaction is waiting for some transaction to finish. A "Blocking set" is defined as the set of all non-blocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T [Ref. 5]. A pair (T,T') is a "blocking pair" of T if T' is in the blocking set of T. A "Potential Blocking set" consists of all waiting transactions that can be reached from T

[Ref. 7]. Sorig(T) means the site of origin of transaction T. Sk is the site currently executing the algorithm. The rules which define the enhanced algorithm, as executed as site Sk, are:

Rule 0: When a transaction T requests a nonlocal resource it is marked "waiting".

Rule 1: The resource R at site Sk cannot be allocated to transaction T because it is held by T1, ...,Tk. Add an arc from T to each of the transactions T1,...,Tk. If there is then a cycle formed in the TWF graph, deadlock has been detected. Otherwise, for each transaction T' in blocking set(T), send the blocking pair (T,T') to Sorig(T) if Sorig(T) =/= Sk and to Sorig(T') if Sorig(T') =/= Sk. Form a list of potential blocking pairs associated with T.

Rule 2: A blocking pair (T,T') is received. Add an arc from T to T' in the TWF graph. If a cycle is formed, then a deadlock exists.

Rule 2.1: If T' is blocked and Sorig(T) =/= Sk, then for each transaction T" in the blocking set(T), send the blocking pair (T,T") to Sorig(T") if Sorig(T") =/= Sk.

Rule 2.2: If T is waiting and Sorig(T) = Sk, then for each potential blocking pair (T",T) send the blocking pair (T",T) to Sorig(T") if Sorig(T") =/= Sk. Then, discard the potential blocking pairs (T",T) and erase the "waiting" mark of T.

Figure 4 shows the actions taken at each site during the execution of the DDA following the request by T4 for resource R3. As can be seen, the deadlock was correctly detected by site A, in absence of failures. If the request message (T4,R3) from site D to site C was lost, however, deadlock detection activity would cease. If the blocking pair (T4,T1) from site C to site D was lost, site A would

still detect the deadlock. If, however, the blocking pair (T4,T1) from site C to site A was lost, site D would apply rule 2. Neither rule 2.1 or 2.2 applies, so deadlock detection activity would cease.

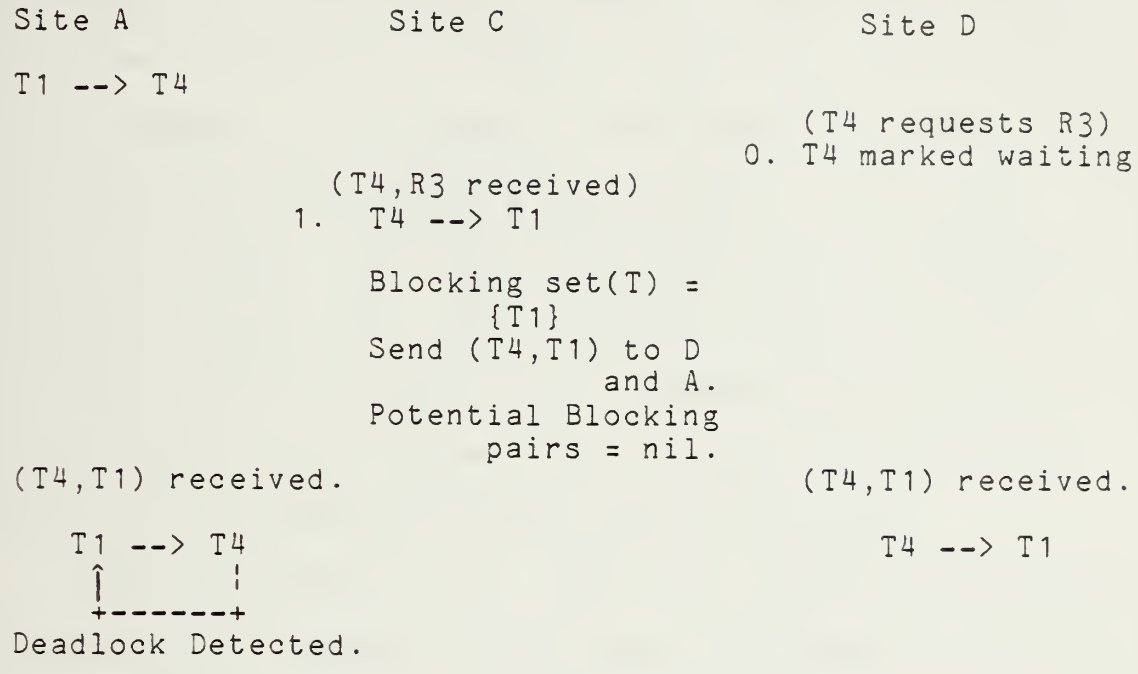


Fig. 4 -- Execution of the Menasce-Muntz DDA

This algorithm allows sites of types b, c, d and e, although this example does not include a site of type e. If a type b site (one having a transaction involved in the deadlock and the site is also involved in detection) failed, in this example site A (or site D), the behavior of the algorithm is dependent on the time of failure. If site A failed before receiving the blocking pair (T4,T1), site C would recognize the failure, but its action is not specified in the rules of the DDA. Site D would not detect the

deadlock for the same reason as if the message from site C to site A was lost. If, however, the failure occured after site A received the blocking pair, deadlock detection activity would continue (at site D) but deadlock would not be detected. A failure of site D, also a type b site, at any time, would have no effect on detecting the deadlock in this example. If a type c site failed (site B), it would have no effect on detecting the deadlock. If a type d site (site c) failed, the time of its failure would determine the behavior of the DDA. If it failed before sending the blocking pair to sites A and D, deadlock detection activity would cease. If it failed after sending those messages, it would have no effect on detecting the deadlock.

For this example, this algorithm behaved surprisingly similarly to Goldman's algorithm in almost all types and timings of failures. This may just be an anomaly found in small deadlock cycles, because in longer and more complex scenarios, it would appear that more sites would be involved in detection, and that there would be some duplication of information. As the number of transactions (and resources) involved in a deadlock cycle increases, more blocking pairs and potential blocking pairs will be sent to more sites, i.e., the number of sites detecting the deadlock increases with the number of transactions involved in the deadlock and with the deadlock topology (or complexity). Thus there will be more chance of a deadlock being detected, as more

parallel detection activity will be in progress. It appears, then, that as the site and complexity of deadlock increases, the robustness of this algorithm increases. However, as pointed out by Gligor and Shattuck, the effect of rule 2.2 discarding information too early may have some impact on the increased robustness.

C. THE ALGORITHM OF OBERMARCK

Obermarck's distributed algorithm [Ref. 1] constructs a transaction-waits-for (TWF) graph at each site. Each site conducts deadlock detection simultaneously, passing information to one other site. Deadlock detection activity at a site may become temporarily inactive until receipt of new information from another site. Obermarck states that in actual practice, synchronization (not necessarily precise) between sites would be roughly controlled by an agreed-upon interval between deadlock detection iterations, and by timestamps on transmitted messages. Nodes in the graph represent transactions, and edges represent a transaction-waits-for-transaction (TWFT) situation. A "String" is a list of TWFT information which is sent from one site to one or more sites. In the model of transaction execution used by Obermarck, a transaction may migrate from site to site, in which case an "agent" represents the transaction at the new site(s). A communication link is also established between agents of a transaction. These communication links are

represented by a node called "External." An agent which is expected to send a message is shown in the WF graph by EX-->T, while an agent waiting to receive is shown by T-->EX. Although Obermarck's algorithm includes the resolution of deadlocks, only the detection part will be considered in this analysis. Transaction ID's are network unique names for transactions, and are lexically ordered. (For example, T1 < T2 < T3). The steps performed at each site are:

1. Build a TWF graph using transaction to transaction wait-for relationships.

2. Obtain and add to the existing TWF graph any "strings" transmitted from other sites.

   a. For each transaction identified in a string, create a node in the TWF if none exists in this site.

   b. For each transaction in the string, starting with the first (which is always "external"), create an edge to the node representing the next transaction in the string.

3. Create wait-for edges from "external" to each node representing a transaction's agent which is expected to send on a communication link.

4. Create a WF edge from each node representing a transaction's agent which is waiting to receive from a communication link, to "external."

5. Analyze the graph for cycles.

6. After resolving all cycles not involving "external", if the transaction ID of the node for which "external" waits is greater than the Transaction ID of the node waiting for "external", then

   a. Transform the cycle into a string which starts with "external", followed by each transaction ID

in the cycle, ending with the transaction ID of the node waiting for "external".

b. Send the string to each site for which the transaction terminating the string is waiting to receive.

In his proof of correctness, Obermarck shows how the algorithm can detect false deadlocks because a string received at a site may no longer be valid when it is used. He discusses two methods of handling false deadlocks; treat them as actual deadlocks(if they don't occur too often), or verify them by sending them around the network and have each site verify them.

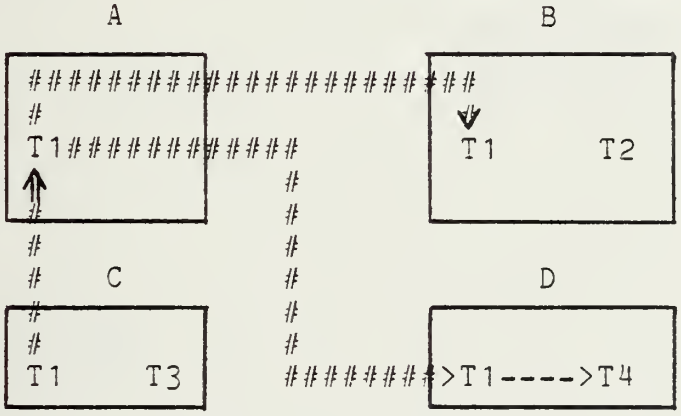


Fig. 5 -- Initial conditions for the Obermarck DDA

Figure 5 shows a global picture of the system, including the communication links established between agents, for the initial conditions of this example. The agents of T1 at sites B and C have performed work (used R2 and R3), and are waiting for the next request from T1 at site A. T1 at site A is waiting for its agent at site D, which is in resource

wait for T4. Figure 6 shows the actions of this algorithm in an environment of no errors. As can be seen, it successfully detects the deadlock.

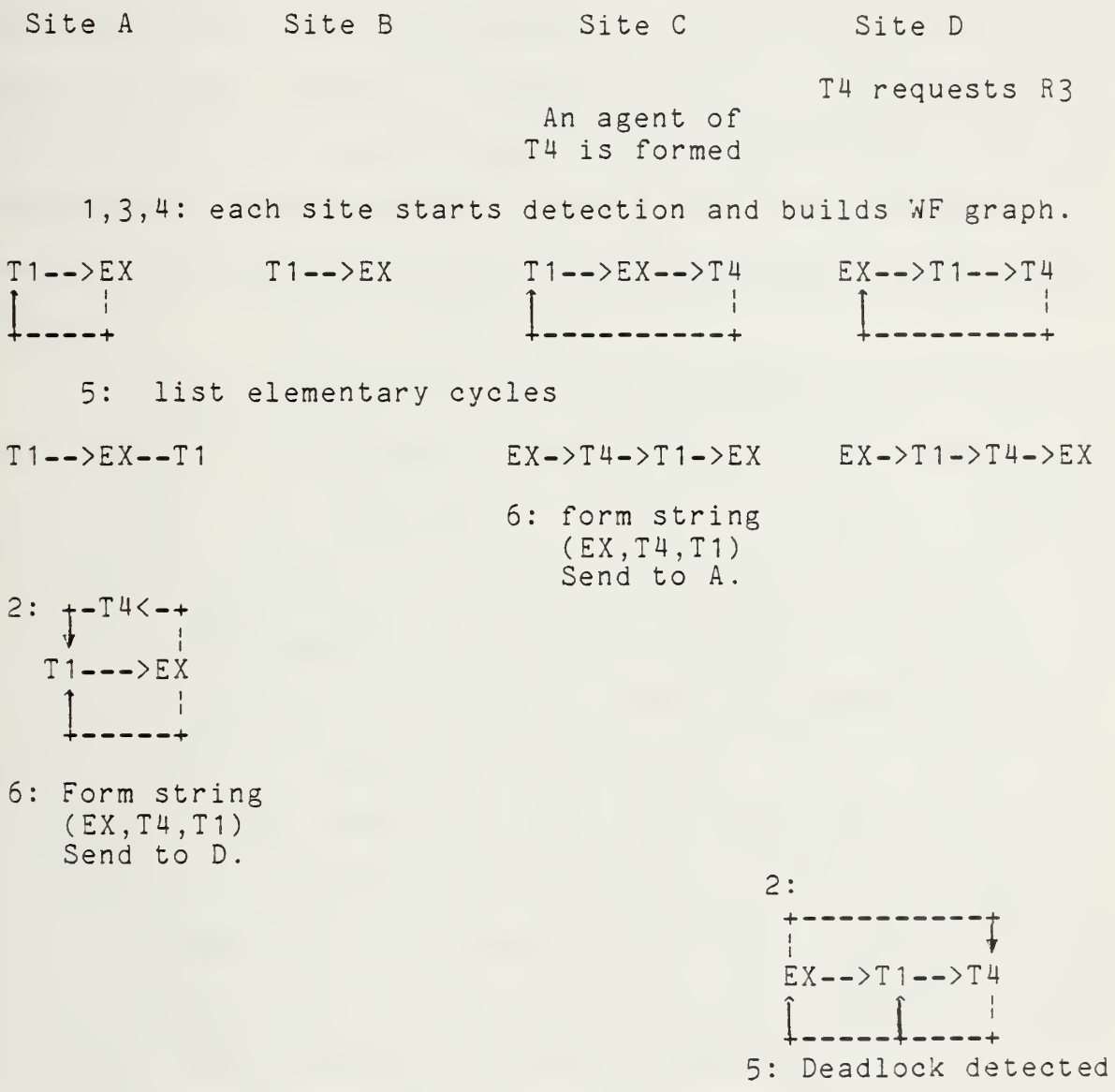


Fig. 6 -- Execution of the Obermarck DDA

Obermarck assumes that messages sent are received. This is essential to the correctness of this DDA, because it is

easy to see what happens if a message is lost. If the string (EX,T4,T1) from site C to A, or from A to D were lost, deadlock detection activity would cease without detecting the deadlock. The use of agents to represent transactions which have migrated to other sites allow this DDA to have nodes of types a or b, if 'agents' is substituted for 'transactions' in the definitions at the beginning of this section. Site B would be an example of a type a site, while the other three sites would all be type b sites.

A failure in site B would have no effect on the behavior of the DDA. A failure at sites A, B or D would either have no effect, an undetermined effect, or cause deadlock detection activity to cease, depending on the time of the failure. For example, if site C failed before sending the string (EX,T4,T1) to site A, deadlock detection activity would cease. If site A (or D) failed before the string (EX,T4,T1) was sent to them, the transmitting site would recognize the failure, but its action in that eventuality is not included in the steps of the DDA. If site C failed after sending the string, the detection activity would continue, and the deadlock would be detected.

This DDA appears to be potentially more robust than the previous two. Each site contains and retains more information in its WF graph, and all sites start detection activity simultaneously, and potentially stay involved for

the entire detection process. The use of the lexical ordering of nodes was for optimization of the number of messages transmitted. If this constraint were lifted, the strings would be sent to all sites involved from all sites in which a cycle existed. In this example, this would have allowed sites A and D to simultaneously detect deadlock. The DDA would be clearly more robust, but the overhead would be greater. In its existing form, this DDA's robustness is similar to the previous algorithms because it is essentially sequentially detecting the deadlock.

D. THE ALGORITHM OF TSAI AND BELFORD

In [Ref. 8], Tsai and Belford present a distributed deadlock detection algorithm. They utilize a "Reduced Transaction-Resource" (RTR) graph, which contains only a subset of the transaction resource graph, but has all relevent TWF edges. Nodes in the RTR graph can be transactions or resources. The algorithm uses a concept the authors call a "reaching pair", which is the basic unit of information passed from site to site. If a path TiTj...Tn can be formed by following TWF edges, and if there is a request edge (Tn,Rm), then Ti "reaches" Rm, and (Ti,Rm) is a "reaching pair." Five types of messages are sent between sites: reaching messages, nonlocal request messages, allocation messages, release-request messages, and releasing messages. The non-local request messages include a list of

all resources currently held by the requesting transaction. Five different types of edges are distinquished in the RTR graph: requesting edges, allocation edges, TWF edges, resource reaching edges and transaction reaching edges. A global timestamp is also used to establish an ordering of events. This timestamp is used on allocation, request and reaching messages, and on allocation and reaching edges in the RTR graph. The notation used in the algorithm is:

TS(M): timestamp of a message
TS(C): current system time
TS(A): timestamp of an allocation edge
TS(R): timestamp of a reaching edge
Sorig: Site of origin
=/= : not equal to

The steps of the algorithm (as executed at site Sk) are:

Step 1: {A transaction T enters the system requesting a nonlocal resource R} Add request edge (T,R) to RTR graph. Send request message (T,R',R,TS) to Sorig(R), where R' is the set of all resources allocated to T, and TS(M) = TS(C). R' has each TS(A) attached, and R' is empty if T holds no resources.

Step 1a: {A transaction T releases a nonlocal resource R} Erase edge (R,T) in the RTR graph. Send a release-request message(R,T) to Sorig(R).

Step 2: {A transaction T enters system requesting local resource R} Go to step 4.

Step 2a: {A transaction T releases a local resource R} Erase edge(R,T) in RTR graph. If there is any transaction T' waiting for R, then begin
Add allocation edge (R,T') to RTR graph with TS(A) = TS(C). Send allocation message (R,T',TS) with TS(M) = TS(C) to Sorig(T') if Sorig(T') =/= Sk. end.

Step 3: {A request message (T,R',R,TS) is received} Add allocation edges (Ri,T) for each Ri in R' to RTR graph. Go to step 4.

Step 3a: {A release-request message (R,T) is received} Erase allocation edge (R,T) in RTR graph. Send releasing message (R,T) to Sorig(T). If there is any transaction T' waiting for R, then begin
Add allocation edge (R,T') to RTR graph with TS(A) = TS(C). Send allocation message (R,T',TS) to Sorig(T') if Sorig(T') =/= Sk. end.

Step 4: If R is not held by any transaction, then begin
Add allocation edge (R,T) with TS(A)=TS(C) to RTR graph. If Sorig(T) =/= Sk, then send an allcation message (R,T,TS) with TS(M)=TS(C) to Sorig(T). end.
else begin
Add requesting edge (T,R) to RTR graph. Suppose R is held by transaction T'. Add edge (T,T') to RTR graph. If there is a cycle, deadlock has been detected, else go to step 5. end.

Step 5: {reaching message generation step} If there are two edges (T,R) and (T,T') added to the graph, and if TT'...T" is any path obtained by following the TWF and transaction reaching edges, then set X =R" if T" has outgoing edge to R", else set X = R. For all transaction Ti in RTR graph reaching X via T, do begin
If Ti holds any resource R' with Sorig(Ti) =/= Sorig(R') and Sorig(R') =/= Sk, then send a reaching message (Ti,X,TS) to Sorig(R'). If Sorig(Ti) =/= Sk and Ti =/= T, then send a reaching message (Ti,X,TS) to Sorig(Ti). If Sorig(Ti) =/= Sk and Ti = T and X = R" then send a reaching message (Ti,X,TS) to Sorig(Ti). The TS in the reaching message is set to TS(C) if triggered by a local request, and set to TS(M) of the nonlocal request or reaching message otherwise.

Step 6: {An allocation message (R,T,TS) is received} If R is an entry in the graph, then begin
Erase allocation edge (R,T') and all reaching edges (T",R) with TS(R) < TS(M) and the corresponding TWF edge (T,T') and transaction reaching edges (T",T'), if they exist, where T' =/= T. Change requesting edge (T,R) to

allocation edge (R,T) with TS(A) = TS(M) if (T,R) exists, and for each resource reaching edge (T",R), add the transaction reaching edge (T",T). If Sorig(T) = Sk, wake up transaction T. end.

Step 6a: {A releasing message (R,T) is received} If Sorig(T) = Sk, wake up transaction T.

Step 7: {A reaching message (T,R,TS) is received} If there exists an allocation edge (R,T') in the graph with TS(M) < TS(A) and T' =/= T, then skip this step, else begin

Add resource reaching edge (T,R) to the RTR graph. If R is held by transaction T', then add the transaction reaching edge (T,T') to the graph. If there is a cycle in the graph, there is deadlock (go to step 8), otherwise go to step 5. end.

Step 8: {a deadlock has been detected} Take appropriate action.

Figure 7 shows the starting WF graphs and the actions of the DDA resulting from the request by transaction T4 for resource R3. An important item to note is that as soon the request is made, step 1 adds sufficient information to the WF graph to detect a deadlock, but does not check for deadlock, so the request is sent to site C and the algorithm continues. The obvious thing to do would be to add a check for a deadlock cycle in step one, but on closer analysis, this check may lead to detection of false deadlocks (if, for example, T1 had just released R3 but the message had not yet been received by site D.) Therefore the algorithm in its present form will be analyzed. The only message sent by this algorithm in this example is the request message

(T4,{R4},R3,t6). If it was lost, the current algorithm would cease detection activity without detecting deadlock. In this instance, if the algorithm checked for deadlock in step 1, it would have been detected with no messages required.

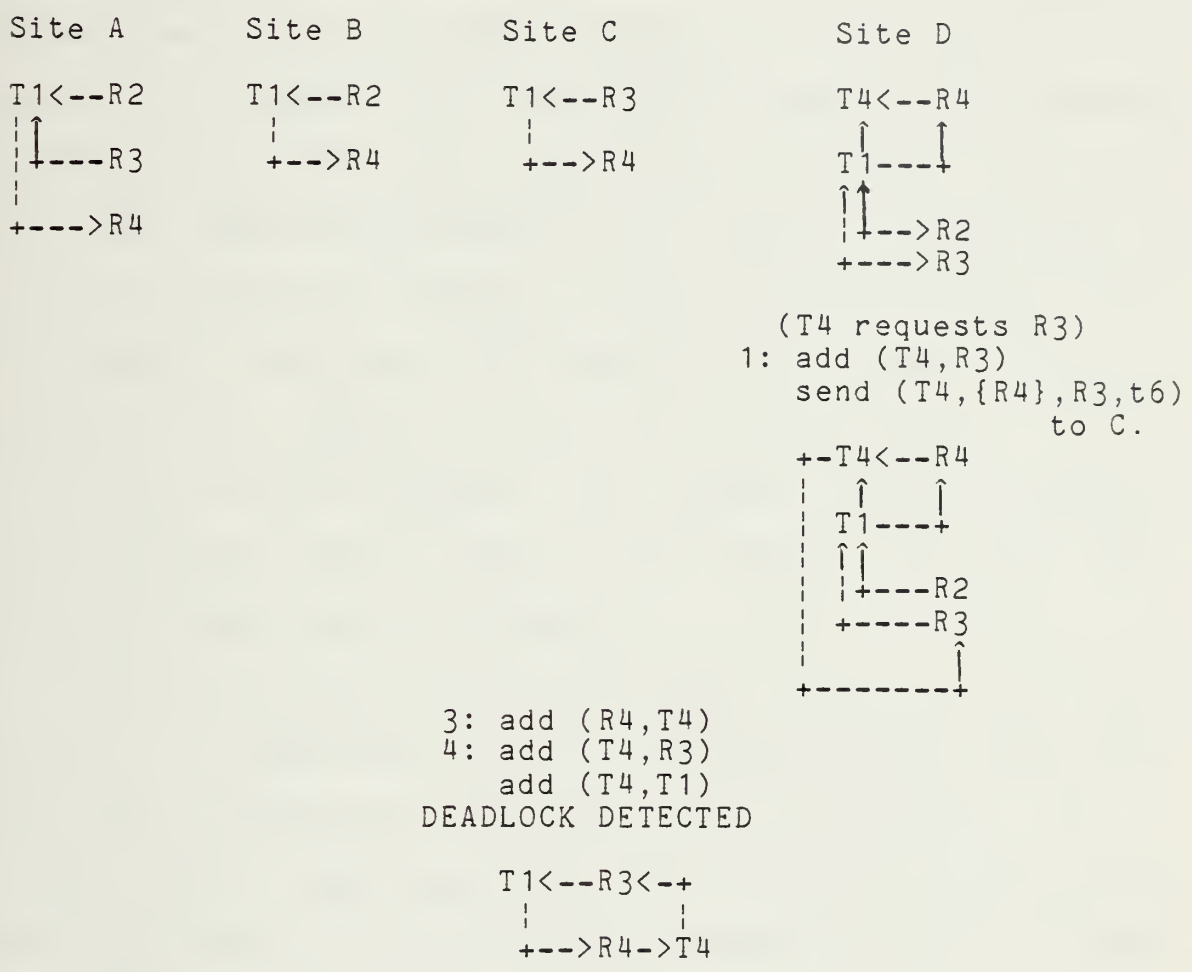


Fig. 7 -- Execution of the Tsai-Belford DDA

For this DDA, sites can be of type b, d or e. Sites A and D are type b and sites B and C are type d. This example has no type e sites, but for other examples, step 5 of the algorithm could send reaching messages to sites not involved at all. Those sites would execute a step or two of the

algorithm, but not be intimately involved in the actual deadlock detection. In this example, a failure of sites A or B (types b and d respectively) would have no effect on the detection of the deadlock. The effect of a failure of site C before the reaching message was sent to it cannot be determined because the DDA includes no instructions for that event. A failure of site C after receiving the reaching message would result in a cessation of detection activity. If the algorithm were modified to include a cycle check in step 1, a failure of site C at any time would have no effect on deadlock detection. The timing of the failure would also determine the behavior of the DDA if site D failed. If site D failed before sending the request message, detection activity would cease, while if the message had been sent, deadlock would still be detected.

For this example, this DDA appears to be about the same level of robustness as the other algorithms, except that each site contains and retains more information than in other DDA's. This indicates that it should be more robust. The algorithm in the case of this example was able to detect the deadlock with only the resource request message. As deadlock cycles become more complex, it appears that this algorithm will also become more robust, even more so than Obermarck's, because this DDA retains more information, and it will send reaching messages to any site potentially involved in the deadlock. Detection activity will occur

simultaneously in those sites receiving reaching messages. The impact of the inclusion of a cycle detection in step 1 may have adverse effects on the correctness, but it might greatly enhance the robustness of the DDA.

E. CONCLUSIONS

The algorithms discussed in the previous section can be loosely ranked by their robustness. Goldman's algorithm is the least robust, because it is always executed sequentially (unless the requests occur simultaneously, as discussed previously). Thus it is always dependent on a single node. Obermarck's algorithm starts deadlock detection simultaneously at all sites, and subsequently passes information in a lexical manner because of the message optimization. For the example used for the analysis, this resulted in a sequential detection, although for larger deadlock cycles, it may have some parallel detection activity occuring. The Menasce-Muntz algorithm starts detection at the site where the deadlock occured, and deadlock detection is subsequently conducted at sites which are potentially involved. In the Tsai-Belford algorithm, deadlock detection can occur simultaneously at all sites potentially involved in the cycle. It appears more robust than the Menasce-Muntz algorithm because more information is held at each site.

The above analysis supports the rather obvious conclusion that a deadlock detection algorithm's robustness is directly related to its cost. The Tsai-Belford algorithm appears more robust than Obermarck's algorithm, for example, but it maintains larger WF graphs at each site, and is invoked each time a resource is requested, in order that the WF graphs contain sufficient information.

For the example used to analyze the four algorithms in Chapter three, the behavior of each of those algorithms in the presence of errors is almost identical. Because the deadlock cycle only involved two transactions, those algorithms which are potentially more robust in the presence of larger cycles, did not have time to demonstrate their robustness. In other words, for a short deadlock cycle, all the algorithms converged within approximately the same length of time (two or three iterations.) Short cycles of length two or three are more probable in existing applications, so all the above algorithms are approximately equally robust in current applications. In future applications (information utility programs, for example), however, a much higher probability of more complex deadlock cycles is expected, which will require a more robust DDA. Conversely, however, as the number of transactions (and sites) increases, it will be important to use a minimum cost DDA.

IV. THE PROPOSED ALGORITHM

A. INTRODUCTION

The proposed algorithm assumes conventional distributed two-phase locking and two-phase commit protocols as described in [Ref. 4]. Two types of locks are supported; Exclusive Write(W) and Shared Read(R). These locks, once placed, are held until the transaction commits or aborts. Additionally, there is an Intention Lock (I) which indicates that a transaction wishes to acquire a lock on a resource, either to modify it (IW) or to read it (IR). Intention Locks are placed on a resource when an agent is created at a site of a locked resource which it requires, or when a resource at the same site is requested but is already locked by another transaction. These Intention Locks are not the same as the Intention Modes used by Gray when he discusses hierarchical locks in [Ref. 4]. Gray uses the Intention mode to "tag" ancestors of a resource in a hierarchical set of resources as a means of indicating that locking is being done on a "finer" level of granularity, and therefore preventing locking on the ancestors of the resource.

A transaction T also modifies its lock entry of the resource it has last locked by adding information that specifies which resource T will attempt to lock next. This modification is made as soon as T can determine which

resource it will require for its next execution step. The rules for locks are the same as for conventional two-phase locking; any number of transactions or agents may simultaneously hold Shared Read locks on a particular resource, but only a single transaction or agent may hold an Exclusive Write lock on a resource. Any number of Intention Locks (IW or IR) may be placed on a resource, which means that any number of transactions may wait for a resource. Each site must therefore have some method for determining which transaction will be given the resource when it becomes free, such as FIFO (First In, First Out.)

The locks can be of any granularity. It must be remembered that a very small granularity (for example, individual fields within a record) will result in very few conflicts, but the cost of the additional locks required to lock smaller fields increases. Conversely, a large granularity (possibly complete records) will result in many locking conflicts, but little cost due to the actual locking of resources. The proposed algorithm does not require a specific level of lock granularity.

The Lock History (LH) of a transaction is a record of all types of locks on any resources which have been requested or are held by that transaction. Each resource ID contains a site identifier. An example of a Lock History for transaction T1 is LH(T1): {W(R3C), W(R2B), R(R1A)}. This LH shows that T1 holds a Write Lock on resource R3 at site C, a

Write Lock on resource R2 at site B, and a Read Lock on resource R1 at site A.

The information contained in a Lock Table for a resource includes a) the transaction or agent ID and its Lock History, b) the type of lock and c) the resource (and type of lock) which that transaction holding this lock intends to lock next. The field containing the current lock will be referred to as the "current" field of the Lock Table, and the field containing the future intentions of that transaction holding the "current" lock will be called the "Next" field. For clarity, Lock Histories will be shown as separate entities. An example of a Lock Table is LT(R2B): T1{W(R2B), IW(R3C)}; T2{IW(R2B)}. The Lock Table for resource R2 at site B shows that T1 holds a Write Lock on R2, and that T2 has placed an Intention Write Lock on R2. T1 has also indicated that it intends to place a Write Lock on resource R3 at site C.

The proposed algorithm also assumes a distributed model of transaction execution where each transaction has a Site of Origin (Sorig), which is the site at which it entered the system. Whenever a transaction requires a remote resource, (a resource at a site other than the site it is currently at), it "migrates" to the site where that resource is located. Migration consists of creating an "agent" at the new site. The transaction agent then executes, and may either create additional agents, start commit or abort

actions, or return execution to the site from which it migrated. This transaction model is consistent with recent literature [Ref. 1, 2]. When a transaction migrates, it brings with it certain information from its previous site. This includes its Lock History and a condensed version of that site's latest Wait-For Graph, which will be termed a Wait-For String (WFS).

A Wait-For Graph (WFG) is constructed by the deadllock detection algorithm, using the Lock Histories of transactions which are possibly involved in a deadlock cycle, any time a transaction or agent attempts to place a lock on a resource which is already locked, or when it determines that a remote resource will be required. There are two types of nodes in the WFG; transactions (or agents) and resources. A directed arc from a resource node to a transaction node indicates that the transaction has a lock on the resource, while a directed arc from a transaction node to a resource indicates that the transaction has placed an Intention Lock on that resource. A directed arc from a transaction node to another transaction node indicates that the first transaction is waiting for the second transaction to release a lock on a resource.

The WFS is a list of transaction - waits - for - transaction strings (obtained from the site's WFG), in which each transaction is waiting for the next transaction in the string, and the Lock History for each transaction in the

string. For example, the WFS [T1{W(R2A), IW(R3B)}, T4{W(R3B)}] shows that T1 is waiting for T4, and each transaction's Lock History is in brackets. A transaction may also bring along other information such as a metric representing its execution cost, but such information is not included in this thesis as it is outside the primary function of the proposed deadlock detector. Each transaction or agent will have a globally unique identifier which indicates its Site of Origin.

Agents can be in any of three states; active, blocked (waiting), or inactive. An inactive agent is one which has done work at a site and created an agent at another site or returned execution to its creating site, and is now awaiting further instructions, such as commit, abort or become active again. A blocked transaction is one which has requested a resource which is locked by another transaction. An active agent is one which is not blocked or inactive. To allow concurrent execution, a transaction may have several active agents.

Each site in the system has a distributed deadlock detector, which performs deadlock detection for transactions or agents at that site. Several sites can simultaneously be working on detection of any potential deadlock cycle.

The basic premise of the proposed algorithm is to detect deadlock cycles with the least possible delay and number of inter-site messages. Based on the findings by Gray and

others [Ref. 9] that cycles of length 2 occur much more frequently that cycles of length 3, and cycles of length 3 occur much more frequently that cycles of length 4, and so on, the proposed algorithm uses a staged approach to deadlock detection. There are basically two types of deadlock cycles to be considered; a) those which can be detected using only the information available at a site, and b) those which require inter-site messages to detect. In the proposed algorithm, the first type has been divided into two levels of detection activity. Because the proposed algorithm checks for possible deadlock cycles every time a remote resource is requested or a local resource is requested but already locked, the level one check should be as quick as possible. If the requested resource is still not available "after X units of time" [Ref. 4], then the probability of a deadlock has increased sufficiently to justify a more complex and time-consuming check in level two. Therefore the proposed algorithm has three levels of deadlock detection activity. Levels one and two correspond to the first type of deadlock cycle, while level three corresponds to the second type. The first level is designed to detect cycles of length 2, although certain more complex deadlock cycles could be detected, depending on the topology of the deadlock cycle. This level uses only information available in the Lock Table of the requested resource if the resource is local, or the last locked resource if the

requested resource is at another site, and in the transaction Lock Histories of the transactions in that Lock Table at the site. Due to the information contained in the "Next" field of the Lock Table and in each transaction's Lock History, this level of detection activity can detect deadlock cycles of length 2 (and possibly longer) involving one or two sites.

As an example, let transaction T1 at site A Write Lock resource R1. Let transaction T2 at site B Write Lock resource R2. These locks would be placed in the Lock Tables of the respective resources, and also in the Lock Histories for the respective transactions. Transaction T1 now determines that it must lock resource R2, so it places that information in the "Next" field of its lock entry of resource R1 and in its Lock History. It then migrates to site B, where its agent places an Intention lock in the Lock Table for R2, and then becomes blocked, waiting for resource R2 to be released. A level one check is made using the Lock Table of R2, showing no deadlock cycles. Now transaction T2 determines that it requires a Write Lock on resource R1. It places that information in the "Next" field of its lock entry in the Lock Table of R2 and in its Lock History. Before T2 migrates to site A, level one of the deadlock detection algorithm looks at the Lock Table for R2 and notices that T1 is waiting for R2. It therefore combines

the Lock Histories of all transactions holding or requesting locks on R2 (T1 and T2) into a WFG, and detects a deadlock.

In this example, the cost of creating an agent of T2 at site A was saved by a very quick check for cycles of length two. Inasmuch as the majority of deadlocks occurring will be of this length, this simple and inexpensive check will detect the majority of deadlocks as they occur. If, in the example just given, transactions T1 and T2 had simultaneously determined the need for locks at the other site, the initial level one check would not have been performed because no transactions were waiting for those resources. Both transactions would have migrated and placed Intention Locks at the new sites. A level one check is then made at each site when it is noted that the requested resource is not available. Each site constructs a WFG from the Lock Histories of the transactions in the Lock Tables of the requested resources, and each site will detect a deadlock cycle in the WFG without any inter-site messages.

Even if the first level of detection activity fails to detect a deadlock cycle, there can still be a more complex deadlock cycle in existence. The second level of detection activity requires more time because it constructs a WFG using all Lock information available at the site, i.e., Lock information from all resource Lock Tables at the site. If we assume that more complex deadlock cycles are comparatively rare, it is advantageous to "wait X units of

time" [Ref. 4] before starting the second level of detection activity. If a transaction is still waiting to acquire a lock after these X units of time, the probability of a more complex deadlock cycle existing has increased sufficiently to justify a more comprehensive check. As previously mentioned, the second level still attempts to detect a cycle using information available at the same site where the transaction is waiting for a resource. Each site maintains the latest WFS brought from each site by transactions which have migrated to that site. In addition, each transaction has a copy of its Lock History. The Lock Histories of all blocked or inactive transactions at the site, and the Lock Histories from all transactions in the WFSs from other sites are combined into a new Wait-For Graph. If no deadlock is detected, it can either be because a) there is no deadlock, or b) there is a deadlock but this site does not have enough information to detect it. Case a) can occur either if all transactions being waited for are currently at that site and active, or have migrated but are still active. If all the transactions being waited for are currently at that site and active, deadlock detection activity can stop, because there can be no cycle in the WFG. If, however, a transaction has migrated to another site and therefore the current site does not have sufficient information to detect whether that transaction is active or blocked, this site's information must be shared with other

sites to determine if the transaction which has migrated is active or is blocked and involved in a deadlock cycle. This sharing of information constitutes the third level of detection activity.

Because level three involves inter-site communication, it might be advantageous to wait Y units of time before continuing in order to increase the probability of the wait condition being an actual deadlock. After Y units of time, when the DDA is ready to continue, the WFG is condensed into a WFS. The WFS is then sent to other sites. The sites to which the WFS is sent can vary. In the version presented here, it is sent to the site to which the transaction being waited for has migrated. Other possibilities are discussed in Chapter six. When a site receives a WFS, it substitutes the latest Lock Histories for any transaction for which it has a later version. It then constructs a new WFG and checks for cycles. If a cycle is found, it must be resolved. If any transactions are waiting for other transactions which have migrated to other sites, the current site must repeat the process of constructing WFG's and sending them to the sites to which the transactions being waited for have migrated. If the transactions being waited for are at this site and active, deadlock detection activity can cease. Level three activity will continue until a deadlock is found or it is discovered that there is no deadlock.

The following definitions are used in the description of the algorithm:

IL	--	Intention Lock
W(x)	--	Exclusive Write lock on resource x
R(x)	--	Shared Read lock on resource x
IW(x)	--	Intention Lock(Write) on resource x
IR(x)	--	Intention Lock(Read) on resource x
Sorig(T)	--	Site or Origin of transaction T
LT(R)	--	Lock Table for resource R
LH(T)	--	Lock History for transaction T
"Next"	--	Field in Lock Table reflecting the resource a transaction intends to acquire next
"Current"	--	Field in Lock Table reflecting the lock currently held by a transaction

B. THE ALGORITHM

1. {Remote resource R requested or anticipated by transaction or agent T}

   A. Place appropriate IL entry in "next" field of the Lock Table of the current resource (the last resource locked by T,if any) and in LH(T).

   B. {Start level 1 detection activity at current site}. If another transaction is waiting for the last resource locked by T, construct a Wait-For graph from the Lock Histories of the transactions holding and requesting that resource and check for cycles.

   C. If no cycles are detected or if no transactions are waiting:

      1) Collect LH(T) and the latest WFS from the current site, and have an agent created at the site of the requested resource.

      2) Stop

   D. If a cycle is detected, resolve the deadlock

2. {Local resource R requested}

A. If resource R is available: {Lock it}

1) Place appropriate lock in Lock Table of resource R and in LH(T).

2) end

B. If resource is not available: {Start level 1 detection activity}

1) Place appropriate IL in Lock Table of resource R and in LH(T).

2) Construct a WF Graph from Lock Histories of all transactions holding and requesting R, and check for cycles.

3) If there are no cycles, and if the transaction holding the lock on R is still at this site and active, stop. If there is a cycle, resolve the deadlock.

4) If the transaction holding the lock on R has either migrated to another site, or is still at this site but is blocked by another transaction which has migrated to another site, delay(t1).

5) If resource is now available:

a) Remove IL from Lock Table and LH(T)

b) Go to step 2A

6) If resource is not available: {Start level 2 activity}

a) Construct a WFG using the Lock Histories of the transactions in the WFSs which have been sent from other sites by level three detection activity, or brought by transactions which have migrated to this site, and the Lock Histories of all blocked or inactive transactions at this site and check for cycles.

b) If any cycles are found, resolve the deadlock.

c) If no cycles are found, Delay(t2)

d) If the requested resource is now available, go to step 2A

e) If the transaction being waited for is at this site and active, stop.

f) If the resource is still not available, go to step 3 {Start level 3 detection activity}.

3. {Wait-For Message Generation}

A. {Start Level 3 detection activity} Construct a WFS by condensing the latest WFG into a list of strings of transactions waiting for transactions. Add the Lock Histories of each transaction in the string.

B. Send the WFS to the site to which the transaction being waited for has gone.

4. {Wait-For Message Received}

A. {Start level 3 detection activity} Construct a WFG from the Lock Histories of the transactions in the WFS's from other sites, and from the Lock Histories of all blocked or inactive transactions at this site. (Use the latest WFS from each site.)

B. If this WFG shows that a transaction which is being waited for has migrated to another site, go to step 3. {Repeat WFS Generation}

C. If the transaction being waited for is active, and has not indicated by an Intention Lock that it will attempt to acquire a resource which may result in a deadlock, discard the WFG and stop.

D. If the transaction being waited for is active but has indicated by an Intention Lock that it is going to a site which will cause a deadlock, or if a cycle is found, resolve the deadlock.

C. EXPLANATION OF THE ALGORITHM

Step 1. This step is executed any time a transaction (or agent) T requests a remote resource, or when it determines that it will require a remote resource. The Lock

Table of the resource which the transaction is currently using (or has just finished with) is checked to see if any other transactions are waiting (i.e., have placed Intention Locks) for that resource. If so, the Lock Histories of all transactions requesting and holding the resource are combined into a WFG and a check for cycles is made. If no cycle is found, T collects the WFS formed from the WFG at that site and causes an agent to be created at the site of the requested resource.

Step 2. This step is executed each time a local resource is requested, either by an agent (transaction) already at that site or by a newly created agent. If the resource is available, appropriate locks are placed and the resource granted. If the resource is not available, Intention Locks are placed in the Lock Table of the requested resource and in the Lock History of the requesting transaction, a WFG is constructed using only the information in the Lock Table of the requested resource and the Lock Histories of the transactions holding or requesting that resource, and a quick level one check is made for possible deadlock cycles. If no cycles are found, the algorithm waits for a certain period of time before continuing. This should allow the transaction which holds the resource to complete its work and release the resource. If the resource is not available after this delay, the chance of a deadlock is higher, so the algorithm shifts to another level of

detection. It now uses the Lock Histories from each blocked or inactive transaction at the site, as well as from any WFS's from other sites which have been brought by migrating transactions. If there are no cycles in this graph, and the resource is still not available after a second delay (also tunable by the system users), the possibility of deadlock is again much greater, but the current site has insufficient information to detect it. Therefore the proposed algorithm progresses to the third level of detection (step 3).

Step 3. The Wait-For message generated by this step consists of a collection of substrings. Each substring is a list of transactions each of which is waiting for the next transaction in the substring. The substring also contains the resources Locked or Intention Locked by each transaction in the substring. A local timestamp will be affixed to this message so that the receiving site will be able to determine which is the latest information from any site.

Step 4. In this step, the Lock Histories of the transactions in the WFS's previously received from other sites, and the Lock Histories of any blocked or inactive transactions at this site are added to the Wait-For information contained in a received WFS. If there is still insufficient information to detect a cycle (a transaction being waited for has migrated to another site), another iteration must be performed, so the algorithm repeats by transferring to step 3. If a cycle is detected, it is

resolved, and if the last transaction being waited for is still active, the algorithm stops.

D. OPERATION OF THE ALGORITHM

The operation of the algorithm will be shown by executing it on the example used in Chapter two. T1 migrates to site B and locks resource R2. It then migrates to site C and locks resource R3. T4 locks resource R4 at site D. At this point, the Lock Histories and Lock Tables are:

Site A	Site B	Site C	Site D
LH(T1): {IW(R2B)}	LH(T1): {W(R2B), IW(R3C)}	LH(T1): {W(R2B), W(R3C)}	LH(T4): {W(R4D)}
	LT(R2B): T1{W(R2B)}	LT(R3C): T1{W(R3C)}	LT(R4D): T4{W(R4D)}

T1 now attempts to acquire resource R4. By step 1, an IL entry is placed in LH(T1) and in LT(R3) at site C. As there are no Intention Locks in LT(R3C), the WFS from site C is collected (at this point in time, none exists), and an agent of T1 is created at site D, with T1 "bringing" LH(T1): {W(R2B), W(R3C), IW(R4D)}. Site D now applies step 2B1, and places the IL entry in LT(R4D) and LH(T1). Then it executes step 2B2 by combining the Lock Histories of T1 and T4. No cycles are found, but as T4 is still active at site D, the DDA is stopped. The current status of the Lock Tables and Lock Histories is:

Site A	Site B	Site C	Site D
LH(T1): {IW(R2B)}	LH(T1): {W(R2B), IW(R3C)}	LH(T1): {W(R2B), W(R2C), IW(R4D)}	LH(T4): {W(R4D)}
			LH(T1): {W(R2B), W(R3C), IW(R4D)}
	LT(R2B): T1{W(R2B)}	LT(R3C): T1{W(R3C), IW(R4D)}	LT(R4D): T4{W(R4D)}; T1{IW(R4D)}

T4 now determines that it needs to write into resource R3. It applies step 1 and places IL entry in LH(T4) and LT(R4D). The Lock Table for R4 is now LT(R4D): T4{W(R4D),IW(R3C)}; T1{IW(R4D)}, and the Lock History for T4 is now LH(T4): {W(R4D), IW(R3C)}. It sees in LT(R4D) that T1 is waiting for R4, so it combines its Lock History with T1's. This reflects the cycle T1-->T4-->T1, so a deadlock has been detected.

V. ANALYSIS OF THE PROPOSED ALGORITHM

The operation of the proposed algorithm was shown in the last chapter. In this chapter, an informal proof of correctness of the algorithm will be presented, and then the algorithm will be analyzed for robustness and efficiency.

A. INFORMAL PROOF OF CORRECTNESS

In general, a deadlock cycle can have many different topologies. For the model of transaction execution used in the proposed algorithm (migration of agents of transactions), these different topologies can be loosely grouped into four categories. Category A involves local deadlocks in which all the resources and transactions involved in the deadlock are local, i.e., located at one site, and thus the transactions involved not have locked any resources at other sites. Category B is the same as category A, with the exception that the transactions are nonlocal, i.e., they may have locked resources at other sites. Deadlock cycles in category C are cycles involving only one transaction and one resource at each of two sites. Category D is a generalization of category C deadlocks; any number of transactions and resources may be involved at any number of sites. For each category, it will be shown that the algorithm detects all possible deadlocks in that

category, and that the algorithm does not detect "false" deadlocks except in the case where a transaction which was involved in a deadlock has aborted, but its agents have not yet been notified. This will be done both for an environment of no errors, and in an environment of the types of errors discussed in Chapter two (lost messages and single site failures.)

If all the transactions and resources involved in a deadlock are located at the same site and none of the transactions have locked resources at other sites, each transaction's Lock History will be an accurate and complete snapshot of the locks placed by that transaction. If the deadlock cycle length is two, the combination of the Lock Histories in step 2B2 (level 1) will detect the cycle. If the length of the cycle is greater than two, step 2B6 (level 2) will combine, for this category of deadlock cycles, the Lock Histories of all the blocked or inactive transactions at the site. This information will be a complete and accurate global snapshot of the deadlock cycle, and hence the deadlock will be detected.

Deadlocks in the second category are those in which all the transactions and resources involved are at one site, but the transactions involved may have locked resources at other sites before creating the agent at this site. The argument to show that all deadlocks in this category will be detected by the proposed algorithm is essentially the same as the one

used for the first category. Since all the transactions involved in the deadlock are currently at this site, their Lock Histories are complete and accurate in so far as they pertain to the deadlock cycle. It is possible, in the case of concurrent execution of a transaction's agents, for an agent involved in a deadlock to be unaware of resources locked by other agents of that transaction which are executing concurrently, and will probably still be active. The only difference between this case and the preceding is that the WFGs constructed by steps 2B2 and 2B6 may contain information about other locks held by the transactions involved, but the information concerning the deadlock cycle will be present.

Deadlocks in the third category will be detected by level 1 because a single Lock Table at each site holds sufficient information to detect a deadlock cycle. If the migrations occur simultaneously, the "Next" field of the Lock Table of the requested resource would show an Intention Lock on the other resource, and this cycle would be detected by step 2B2. If the migrations occurred sequentially, the second transaction would, before migrating, place an Intention Lock in the Lock Table of its last locked resource. The level 1 check of step 1B would cause a WFG to be constructed which would reveal the deadlock cycle.

The fourth category of deadlock cycles is a generalization of the third. Deadlock cycles in this

category may involve any number of transactions and resources at any number of sites. A record is always kept of the site to which a transaction has migrated (in the "Next" field of it's last locked resource at the current site.) If level 2 cannot detect the cycle in step 2B6 with information at that site, level 3 causes a WFS containing this site's information to be sent to the site to which the transaction has migrated. Steps 3 and 4 cause this process to be continued, with each site adding additional information, until a site contains enough information to detect a deadlock cycle or determine that no deadlock exists, regardless of the number of migrations made by a transaction.

False deadlocks are an anomaly where a non-existent deadlock cycle is detected by a deadlock detection algorithm, and are usually a result of incorrect or obsolete information. Since the proposed algorithm uses only the latest copy of a transaction's Lock History for deadlock detection purposes, the information used cannot be incorrect in the sense of invalid entries, although it may be incomplete. This means that a Wait-For graph constructed from incomplete versions of Lock Histories may have insufficient information to detect a deadlock at that particular level of detection activity or iteration of level three activity, but it will not have incorrect information. When a transaction which has agents at two or more sites

commits or aborts, however, it is possible that the commit or abort messages to other agents of that transaction may be delayed. Obviously, a transaction which is ready to commit cannot have any of it's agents in a blocked state (and therefore in a possible deadlock condition), so its agents can either be only active or inactive. While inactive agents may be being waited for by agents of other transactions, no Lock History or Lock Table can show that an agent of the transaction which is about to commit is waiting for another transaction, so no false deadlocks can exist. Therefore only the possibility of a transaction which is in the process of aborting and thus causing a false deadlock to be detected must be considered. Suppose an agent of a transaction decides to abort, but before its abort message reaches another agent of that transaction, a deadlock is found involving that transaction. Technically, this could be considered a false deadlock, since one of the transactions involved has aborted, probably breaking the deadlock cycle. If the deadlock cycle is complex, and the proposed algorithm is performing level two or three detection activity, the delays introduced in steps 2B4 and 2B6c should allow the abort message to arrive. For the very rare occurences where the abort message does not arrive, it would probably be more efficient to let the deadlock detection algorithm resolve the (false) deadlock rather than having the algorithm perform some explicit action (such as

delaying before resolving any detected deadlock cycle) each time it detects a deadlock.

B. ROBUSTNESS ANALYSIS

Level one of the proposed algorithm appears to take a pessimistic view concerning the occurrence of deadlock by checking for it any time a remote resource is requested, or a local resource is not available. The author believes that the cost involved in this simple check is negligible when compared to the cost of creating agents when they are certain to become deadlocked, even when the probability of deadlock is as low as reported in [Ref. 9]. Since cycles of length three or more are very rare, however, it is advantageous to assume an optimistic viewpoint toward their occurence. Thus a greater cost can be expended in checking for them if we wait until the probability of their existence is much higher.

The robustness of the proposed algorithm can be compared to that of the algorithms analyzed in Chapter three by executing it on the example used for that analysis. Additionally, for a more thorough demonstration of the operation of the algorithm, the actions taken by each step will be shown.

At time t1, transaction T1 at site A requests resource R2. Step 1A places an IW (Intention Write) lock in LH(T1). Since T1 currently holds no resources, no Lock Table entries

are made, and step 1B is skipped. Step 1C causes an agent of T1 with LH(T1):IW(R2B) to be created at site B. At site B, step 2 is applied. R2 is available, so the Write Lock is placed in LT(R2B) and in LH(T1) at site B. At time t2, T4 requests R4, which is local. Step 2A is applied by site D, and the Write Lock is placed in LH(T4) and in LT(R4D). At time t3, T1 at site B requests R3. Step 1A places an IW Lock entry in the "Next" field of T1's entry in LT(R2B), and in LH(T1). No other transactions are waiting for R2, so step 1C causes an agent of T1 with LH(T1):W(R2B),IW(R3C) to be created at site C. At site C, R3 is available, so step 2A places the lock in LT(R3C), and modifies the IW(R3C) entry to W(R3C) in the copy of LH(T1) at site C.

At time t4, T1 requests R4. Site C applies step 1A and places an IW(R4D) in the "Next" field of T1's entry in LT(R3C), and in LH(T1). Since no transactions are waiting for R3, site C causes an agent of T1 with LH(T1): W(R2B),W(R3C),IW(R4D) to be created at site D. At site D, R4 is not available, so step 2B places an IW(R4D) entry for T1 in LT(R4D) and in LH(T1). A WFG is constructed from the Lock Histories of T1 and T4. No cycles are detected, and T4 (which has the lock on R4) is still at site D and active, so deadlock detection activity stops. Figure 8 shows the status of appropriate Lock Tables and Lock Histories at all four sites at this time.

Site A	Site B	Site C	Site D
LH(T1): {IW(R2B)}	LH(T1): {W(R2B), IW(R3C)} LT(R2B): T1{W(R2B), IW(R3C)}	LH(T1): {W(R2B), W(R3C), IW(R4D)} LT(R3C): T1{W(R3C), IW(R4D)}	LH(T1): {W(R2B), W(R3C), IW(R4D)} LH(T4):{W(R4D)} LT(R4D): T4{W(R4D)}, T1{IW(R4D)}

Fig. 8 -- Status for proposed algorithm

Now at time t6, T4 requests R3. Site D applies rule 1A by placing an IW entry in LH(T4) and in the "Next" field of T4's entry in LT(R4). It notices that T1 is waiting for R4, so a WFG is constructed using the Lock Histories of T1 and T4. This graph is shown in Figure 9. As can be seen, a deadlock is detected.

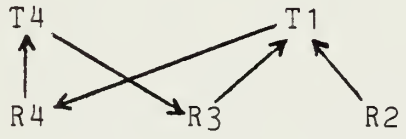


Fig. 9 -- WFG reflecting deadlock cycle

In this example, no messages were used specifically for the deadlock detection, so the effect of a lost message on the DDA cannot be examined. Since three of the previously analyzed algorithms did require separate DDA messages, however, this algorithm is therefore less susceptible to this type of failure. If one of the messages which created an agent were lost, no deadlock would have occured.

Since agents are created at the sites where the required resources are located, and level one detection activity is performed any time a resource is requested, this algorithm allows only sites of type b and d which were discussed in Chapter three. Type b sites are those which have a transaction involved in a deadlock and the site is involved in detection, and type d sites are those which have a resource involved in the deadlock and the site is involved in the detection. In fact, for the version of the proposed algorithm presented in Chapter four, these two types are the same (a site involved in detection will have both a transaction or agent and a resource involved.) The timing and location of a single site failure will determine the behavior of the algorithm. If sites A, B or C failed before creating agents at B, C and D respectively, no deadlock would result. If they failed after having an agent created at the next site, deadlock detection activity would not be affected. If site D failed before an agent for T1 was created, no deadlock would be created. If it failed after creating the agent but before the deadlock was detected, it would not detect the deadlock, but the site failing would in a sense break the deadlock.

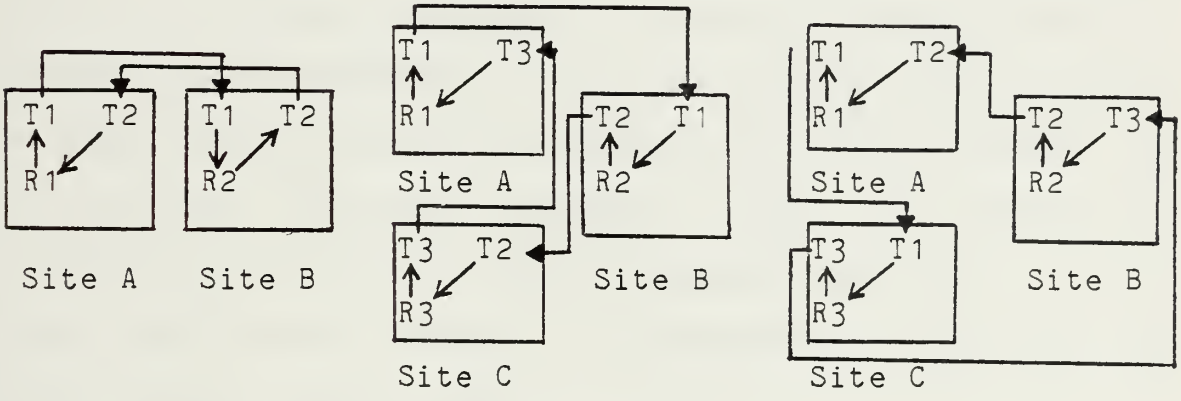
It appears that this version of the proposed algorithm is at least as robust as the Tsai-Belford algorithm, and more robust than the other three algorithms analyzed, for the example used. Because of the three levels of detection

activity in the proposed algorithms, inter-site messages for deadlock detection are only used for deadlock cycles of length three or greater, and depending on the topology of the deadlock cycle, messages are not even required for many of those. In the majority of deadlock occurences, then, even this least robust version of the proposed algorithm appears to be more robust than any of the published algorithms analyzed in Chapter three.

C. PERFORMANCE ANALYSIS

To check the efficiency (in terms of inter-site messages) of the algorithm, it was executed on several deadlock scenarios. The algorithm of Obermarck [Ref. 1] was also executed on these scenarios. Obermarck's algorithm was chosen for this comparison because it is being implemented in IBM's developmental distributed database system, System R*. Since the majority of deadlocks which will occur will be of length two or three, three test cases involving deadlock cycles of those lengths will be used for the comparison. It is assumed that the transactions are lexically ordered T1 < T2 < T3, for Obermarck's algorithm. These cases are shown in Figure 10. T1 originated at site A and holds a lock on R1, and T2 originated at site B and holds a lock on R2. In cases two and three, T3 originated at site C and holds a lock on R3. In case one, T1 has migrated to site B and requested R2, while T2 has migrated

to site A and requested R1. In case two, T1 has migrated to site B and requested R2, T2 has migrated to site C and requested R3, and T3 has migrated to site A and requested R1. In case three, T1 has migrated to site C and requested R3, T2 has migrated to site A and requested R1, and T3 has migrated to site B and requested R2.



Case 1 Case 2 Case 3

Fig. 10 -- Deadlock cycles used in performance analysis

For case one, where the deadlock cycle is of length two, the proposed algorithm requires no additional messages for deadlock detection, while Obermarck's algorithm requires one message. For case two, with a deadlock cycle of length three, Obermarck's algorithm requires one message. The number of messages required by the proposed algorithm is dependent on the timing of the transaction migrations. If the migrations occur at different times, no messages are required. If, however, the migrations happen to occur simultaneously, six messages are required. A similar

situation occurs in case three. If the migrations occur simultaneously, six messages will be required. If they occur at different times, however, no messages are required. Obermarck's algorithm requires two messages.

It is apparent that in the majority of cases (since cycles of length two are more common than those of length three), no messages are required for the proposed algorithm. The worst case scenario for the proposed algorithm, however, is significantly worse than Obermarck's, for the version of the proposed algorithm presented here. If Obermarck's optimization were used with the proposed algorithm, it would never need more messages than Obermarck's algorithm, and would usually require fewer.

The amount of time used in level one activity is minimal, since only a single resource's Lock Table is used to determine the set of transactions whose Lock Histories must be combined. Even with level two, the time required to construct a WFG using all Wait For information at a site should take no longer than the construction of a WFG in Obermarck's algorithm. In [Ref. 1], Obermarck does not discuss the factors which trigger deadlock detection, but for this analysis, it is assumed that it is triggered X units of time after a transaction waits for a resource. His algorithm constructs a WFG at each iteration of the deadlock detection cycle, regardless of the potential size of the cycle. Since the proposed algorithm performs a comparable

construction only when cycles of length two have essentially been eliminated as a possibility, it appears that the proposed algorithm will require less time to execute whenever it is invoked.

VI. CONCLUSIONS

The proposed algorithm has been shown to be at least competitive with existing algorithms for deadlock detection in distributed computing systems. The proposed algorithm is more efficient at detecting the majority of deadlocks which can occur than the other deadlock detection algorithms analyzed. It's performance is worse, however, for those deadlock cycles of length three or greater which are caused by simultaneous migration of all the transactions involved. Inasmuch as deadlock cycles of length three or more are rare, and the probability of all the transactions involved migrating simultaneously appears to be low, this extra cost should be neglibible when compared to the savings caused by the algorithm for the majority of deadlock cycles. If it is felt necessary, an optimization scheme similar to Obermarck's lexical ordering of transactions [Ref. 1] could be included in step 3B, but this requires a global mechanism for ordering all transactions in a system.

The proposed algorithm has been shown to be more robust than the other algorithms analyzed, primarily because it very rarely uses inter-site messages for deadlock detection, and because of the model of transaction execution it assumes. Resources can only be locked by agents at that site, and hence a resource cannot be permanently locked by

virtue of the site where the transaction which holds the lock is located failing. A site failure with this model of transaction execution wil break the deadlock.

The proposed algorithm can be modified by combining levels one and two, if the number of resources and transactions in the system are small, and therefore the cost of creating WFG's at level 2 would be comparable to the cost of the level 1 WFG construction. The cost of construction of the WFG's used by the algorithm could be saved by not constructing them at all, but merely examining the WFS's and Lock Histories, since all required information is contained in them. The delays which have been built-in to the algorithm can be adjusted empirically to determine the optimum delays for a particular implementation.

It is concluded that the proposed algorithm as presented in Chapter four is more robust and efficient than existing deadlock detection algorithms for distributed computing systems, and that its performance can be made even better with minor modifications. It is also a good basis for more major modifications such as having level 3 detection done by the Sites of Origin of the transactions involved.

LIST OF REFERENCES

1. IBM Research Division Research Report RJ2845 (36131), Global Deadlock Detection Algorithm, by R. Obermarck, June 1980.

2. Tandem Technical Report TR81.3, The Transaction Concept: Virtues and Limitations, by J. Gray, June 1981.

3. Massachusetts Institute of Technology Technical Report TR-MIT/LCS/TR-185, Deadlock Detection in Computer Networks, by B. Goldman, September, 1977.

4. IBM Research Division Research Report RJ2188(30001), Notes on Data Base Operating Systems, by J. Gray, February, 1978.

5. Menasce, D. and Muntz, R., "Locking and Deadlock Detection in Distributed Data Bases," IEEE Transactions on Software Engineering, v. SE-5, No. 3, p. 195-202, May 1979.

6. Isloor, S. and Marsland, T., "An Effective 'On-line' Deadlock Detection Technique for Distributed Data Base Management Systems," Proceedings COMSAC, p. 283-288, 1978.

7. Gligor, V. and Shattuck, S., "On Deadlock Detection in Distributed Systems," IEEE Transactions on Software Engineering, v. SE-6, p. 435-440, September 1980.

8. Tsai and Belford, G., "Detecting Deadlock in a Distributed System," Proceedings INFOCOM, 1 April 1982.

9. Gray, J., Homan, P., Korth, H. and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Distributed Database System," paper presented at 5th Berkeley Workshop, February 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Dushan Badal, Code 52Zd Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	William Shockley, Code 52Sp Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	LCDR Michael T. Gehl, USN Box 436 Dow City, Iowa 51528	2
7.	CAPT Peter Jones, USMC Marine Corps Central Design and Programming Activity Marine Corps Development and Education Command Quantico, Virginia 22134	1
8.	CDR Geir Jevne SMC 1675 Naval Postgraduate School Monterey, California 93940	1

Thesis
G258 Gehl
c.1 198112
Deadlock detection in distributed computing systems.